Clase 4 — 21.11.25

#VSC

👩‍🏫 Profesora: Sara Gonzalo
📘 Desarrollo de interfaces. JAVASCRIPT, JQUERY, REALIDAD VIRTUAL
🗓️ Clase 4 — 21/11/2025
🎯 **Tema:** Final Continuacion Java Script | DOM

1️⃣2️⃣ Bucles en JavaScript

Los bucles permiten repetir instrucciones múltiples veces. Sara recalca que en JavaScript funcionan igual que en Java o C, a diferencia de Python.

🔹 Bucle for (cuando sabemos cuántas veces repetir)

```
for (let contador = 0; contador < 10; contador++) {
    // instrucciones
}
```

Características:

- Usa **contador**, **condición** y **actualización**.
- Es el bucle más utilizado cuando el número de iteraciones está claro.
- Estructura idéntica a lenguajes clásicos (Java/C/C++).

Ejemplo visto en clase:

```
for (let contador = 1; contador < 11; contador++) {
    document.write("El valor de contador es: " + contador + "<br>");
}
```

Sara recuerda que `document.write` es solo para practicar, **no se usa en proyectos reales**.

1️⃣3️⃣ Bucle while

El while es **más restrictivo**:

- Solo entra si la condición ya es verdadera.
- Si no se cumple, el bloque **no se ejecuta ni una sola vez**.

```
let num = 0;

while (num != 0) {
    window.alert("Dentro del bucle!!!");
}
```

1️⃣4️⃣ Bucle do…while

Este bucle se creó para resolver la limitación del while:

✔ entra al menos **una vez**
✔ luego evalúa la condición

Se usa cuando **es obligatorio ejecutar algo antes de comprobar**.

```
let num2 = 0;

do {
    window.alert("Dentro del bucle do while");
} while (num2 != 0);
```

1️⃣5️⃣ Arrays en JavaScript

Un **array** es una estructura de datos que permite almacenar **múltiples valores** en una sola variable, organizados en celdas indexadas numéricamente.

Características clave:

- Los índices comienzan en **0** → la primera posición es `0`, la última es `length - 1`.
- Son **dinámicos** → pueden crecer o reducirse durante la ejecución del programa.
- Pueden almacenar **valores de distintos tipos** (algo que no ocurre en lenguajes fuertemente tipados como Java o C#).
- Internamente, un array es un **objeto especial** con propiedades y métodos propios.

Ejemplo:

```
let alumnos = ["pedro", "maria", "eva", "lucia", 23, true, false];
```

El array anterior mezcla:

- strings
- números
- booleanos

Esto es válido porque JavaScript no exige un tipo homogéneo.

Recomendación de Sara (muy importante):

Declarar arrays sin número de posiciones:

```
let tabla = [];
```

Motivos:

- Evita errores si el array crece más de lo previsto.
- Es más flexible cuando los datos provienen de formularios, cálculos o API externas.
- Permite trabajar con arrays “en crudo”, tal y como se usan en la industria.

1️⃣6️⃣ Acceso, modificación y longitud

🔹 Acceder a un valor

Usamos el índice entre corchetes:

```
alumnos[0]   // "pedro"
alumnos[3]   // "lucia"
```

🔹 Modificar un valor

```
alumnos[0] = "Luis";
```

Cuando modificas, **no cambias el tamaño del array**, solo el contenido de esa celda.

🔹 Longitud del array

```
alumnos.length
```

Devuelve la cantidad de celdas existentes.

Sara remarcó el error más común de los estudiantes:

❌ `lenght`
✔️ `length`

Detalle importante:

La propiedad `length` **no es inmutable**.
Si asignas un índice superior, el array crece automáticamente:

```
let datos = [];
datos[5] = "hola";

console.log(datos.length);  // 6
```

Las posiciones intermedias quedan como `undefined`.

1️⃣7️⃣ Métodos de arrays

Los arrays tienen métodos muy útiles para manipular datos:

`push()`

Añade un elemento **al final**.

```
frutas.push("kiwi");
```

`pop()`

Elimina y devuelve el **último elemento**.

```
let ultimo = frutas.pop();
```

`shift()`

Elimina y devuelve el **primer elemento**.

```
frutas.shift();
```

`unshift()`

Añade un elemento **al principio**.

```
frutas.unshift("fresa");
```

Otros métodos los verás cuando lleguemos al DOM y JSON (`splice` , `slice` , `map` , `filter` , etc.), pero Sara se centra por ahora en los más básicos.

1️⃣8️⃣ Recorrer arrays

Recorrer un array significa **visitar cada celda una por una** para leer su valor, modificarlo o usarlo en algún cálculo.

🔹 For clásico (Sara: “el más eficiente y controlado”)

```
for (let i = 0; i < frutas.length; i++) {
    console.log(frutas[i]);
}
```

Ventajas:

- Máximo rendimiento.
- Control completo del índice.
- Recorre exactamente las posiciones válidas.

Es el método estándar para arrays grandes.

🔹 For…in

```
for (let celda in frutas) {
    console.log(celda);  // imprime el índice
}
```

Sara advierte:

- Es **más lento**, porque recorre *propiedades enumerables* del objeto, no solo índices.
- Puede recorrer claves añadidas manualmente.
- No es adecuado para arrays grandes.

Se utiliza más en objetos, no en arrays.

1️⃣9️⃣ Arrays multidimensionales (Matrices)

Una **matriz** es un array cuyos elementos son otros arrays. En JavaScript se usan para representar **datos tabulares** y para **volcar información de fuentes externas** (APIs, JSON, bases de datos).

📌 Cuándo usar matrices

- Tablas (notas, inventarios, horarios).
- Datos estructurados por filas/columnas.
- Respuestas JSON con listas de listas.
- Preparación de datos para renderizar en DOM (tablas HTML).

🔹 Ejemplo 1: Tabla de notas

```
let notas = [
    ["Ana", 7, 8, 9],
    ["Luis", 6, 5, 7],
    ["Eva", 9, 8, 10]
];
```

- `notas[i]` → fila (un alumno).
- `notas[i][0]` → nombre.
- `notas[i][1..n]` → calificaciones.

Recorrido:

```
for (let i = 0; i < notas.length; i++) {
    for (let j = 1; j < notas[i].length; j++) {
        console.log(notas[i][0], "nota:", notas[i][j]);
    }
}
```

🔹 Ejemplo 2: Inventario simple

```
let inventario = [
    ["Teclado", 15, true],
    ["Ratón", 30, false],
    ["Pantalla", 8, true]
];
```

Estructura típica:

- `[nombre, stock, disponible]`

🔹 Ejemplo 3: Matriz desde datos externos (JSON simulado)

```
let datosAPI = [
    [101, "Pedido A", "Enviado"],
    [102, "Pedido B", "Pendiente"],
    [103, "Pedido C", "Cancelado"]
];
```

Este formato es habitual al **consumir APIs** y luego **pintar tablas** con DOM.

🧠 Patrón mental para matrices (muy importante)

- **Primer índice** → estructura principal (fila).
- **Segundo índice** → detalle (columna).
- **Dos bucles** siempre:
  - externo = filas
  - interno = columnas

```
for (let i = 0; i < matriz.length; i++) {
    for (let j = 0; j < matriz[i].length; j++) {
        // matriz[i][j]
    }
}
```

Este patrón se reutiliza **tal cual** cuando trabajes con DOM.

📝 Ejercicios tipo examen

✅ Ejercicio 1

Dada la matriz:

```
let m = [
    [1, 2],
    [3, 4],
    [5, 6]
];
```

Pregunta:
¿Cuántas filas tiene? ¿Cuántos elementos en total?

Respuesta esperada:

- Filas: `m.length` → 3
- Elementos: suma de `m[i].length` → 6

✅ Ejercicio 2

Recorre la matriz y muestra solo los números pares.

```
for (let i = 0; i < m.length; i++) {
    for (let j = 0; j < m[i].length; j++) {
        if (m[i][j] % 2 === 0) {
            console.log(m[i][j]);
        }
    }
}
```

✅ Ejercicio 3 (muy típico)

Suma todos los valores de una matriz.

```
let suma = 0;

for (let i = 0; i < m.length; i++) {
    for (let j = 0; j < m[i].length; j++) {
        suma += m[i][j];
    }
}

console.log(suma);
```

🔗 Conexión directa con el DOM (adelanto)

Sara enlaza este tema con lo siguiente:

- El **DOM es una estructura jerárquica**, igual que una matriz.
- Aprender a recorrer matrices te prepara para:
  - recorrer nodos
  - acceder a hijos
  - crear elementos dinámicos

👉 **Matrices → DOM → Páginas generadas solo con JavaScript**

2️⃣0️⃣ Comentarios finales de Sara

- **while → más restrictivo**, depende completamente de la condición inicial.
- **do…while → garantiza ejecutar al menos una vez**.
- **arrays → declararlos vacíos y llenarlos según el flujo del programa**.
- **matrices → fundamentales cuando consumamos datos externos**.
- Aquí termina la parte de lógica; **en la siguiente clase comenzamos DOM**.

2️⃣1️⃣ Ejercicios de operadores lógicos (tarea obligatoria)

Estos dos ejercicios consolidan el uso de:

- `prompt()` para recoger datos
- Operadores lógicos (`!` , `&&`)
- Condicionales `if / else if / else`
- Validaciones básicas de entrada
- Control de flujo según condiciones encadenadas

Se trabajaron como **tarea obligatoria**, pero encajan perfectamente dentro del contenido de la Clase 4.

🟦 Ejercicio 1 — Uso del operador NOT (`!`)

🎯 Objetivo

- Pedir un número por pantalla.
- Validar que sea **mayor que cero** y **distinto de cero**.
- Si es válido, indicar si es **par** o **impar**.

- Si no es válido, mostrar un mensaje de error.

📌 Código JavaScript comentado

```
// Pedimos un número entero al usuario mediante prompt.
// parseInt convierte el texto introducido en un número entero.
let numero = parseInt(window.prompt("Introduce un número"));

// Validamos que el número sea mayor que cero.
// Usamos el operador lógico NOT (!) para negar la condición "numero <= 0".
// Si "numero <= 0" es true, el NOT lo convierte en false.
// Si "numero <= 0" es false (es decir, número positivo), entonces entra en el if.
if (!(numero <= 0)) {

    // Determinamos si el número es par usando el operador módulo (%).
    // Un número es par si el resto al dividir entre 2 es 0.
    if (numero % 2 === 0) {
        window.alert("El número es PAR");
    }
    // Si no es par, será impar.
    else {
        window.alert("El número es IMPAR");
    }

}
// Si no se cumple la validación principal (mayor y distinto de cero), mostramos
error.
else {
    window.alert("Error: el número debe ser mayor que cero y distinto de cero");
}
```

🧠 Puntos clave reforzados por Sara

- El operador **NOT** es útil cuando queremos negar **condiciones completas**, no valores sueltos.
- Antes de hacer cálculos, siempre validar los datos introducidos por el usuario.
- `prompt()` devuelve texto → siempre convertir a número (`parseInt`, `parseFloat`).

🟦 Ejercicio 2 — Condicionales anidadas y validación por rangos

🎯 Objetivo

1. Pedir un número entero.
2. Comprobar que sea **positivo y distinto de cero**.
3. Si es:
   - **De dos cifras** → indicar si es par o impar.
   - **De tres cifras** → mostrar el resto de dividir entre 2.
4. Si no cumple ninguna condición → error.

📌 Código JavaScript comentado

```
// Solicitamos un número entero al usuario.
let num = parseInt(window.prompt("Introduce un número entero"));

// Primera validación global: el número debe ser positivo y distinto de cero.
// Si no cumple, se muestra error automáticamente.
if (num > 0) {

    // --------- Caso 1: Número de dos cifras ---------
    // Comprobamos si está entre 10 y 99 (ambos incluidos).
    if (num >= 10 && num <= 99) {

        // Determinamos si es par o impar.
        if (num % 2 === 0) {
            window.alert("Número de dos cifras y PAR");
        } else {
            window.alert("Número de dos cifras e IMPAR");
        }
    }

    // --------- Caso 2: Número de tres cifras ---------
    // Entre 100 y 999.
    else if (num >= 100 && num <= 999) {

        // Calculamos el resto de dividir entre 2.
        window.alert("Resto al dividir entre 2: " + (num % 2));
    }

    // --------- Caso 3: No es de dos ni tres cifras ---------
    else {
        window.alert("Error: el número no es de dos ni de tres cifras");
    }

// Si el número no es positivo o es cero, mensaje de error directo.
} else {
    window.alert("Error: el número debe ser positivo y distinto de cero");
}
```

🧠 Conceptos importantes del ejercicio

🔹 Rangos numéricos

Para identificar si un número es de dos o tres cifras:

- Dos cifras → `10 ≤ num ≤ 99`
- Tres cifras → `100 ≤ num ≤ 999`

🔹 Operador AND (`&&`)

Se usa cuando dos condiciones deben cumplirse simultáneamente:

```
num >= 10 && num <= 99
```

🔹 Operador módulo (`%`)

Muy utilizado en programación inicial:

- Para comprobar paridad
- Para obtener restos en divisiones
- Para patrones numéricos

📝 Conclusiones didácticas según Sara

- Estos ejercicios entrenan la **estructura mental** básica para trabajar posteriormente con el **DOM**, donde validar datos será imprescindible.
- Los operadores lógicos se usan para **dirigir el flujo de ejecución** según reglas muy claras.
- Es importante escribir condiciones **legibles**, bien organizadas y sin redundancias.
